Name of Student: Pushkar Sane		
Roll Number: 45		**Practical Number: 8**
Title of Lab Assignment: To implement Threads in Python.		
DOP: 31-10-2023		**DOS: 01-11-2023**
CO Mapped: **CO5**	**PO Mapped:** **PO3, PO5, PSO1, PSO2**	**Signature:**

Practical No. 8

Aim: To implement threads in Python.

1. To do design the program for starting the thread in python.
2. Write a program to illustrate the concept of Synchronization.
3. Write a program for creating a multithreaded priority queue.

Description:

Threading:

Threading is a powerful and essential concept in computer programming that allows for concurrent execution of multiple threads within a single program. Threads are lightweight processes that share the same memory space and can run independently, enabling various functionalities and features. Here are some of the key functionality and features of threading.

Concurrency: Threading provides a way to achieve concurrency in a program. Concurrency allows multiple tasks to make progress and execute independently, leading to improved performance in tasks like I/O-bound operations and responsiveness in applications.

Parallelism: While threading allows for concurrency, it's important to note that due to the Global Interpreter Lock (GIL) in Python, true parallelism in CPU-bound operations is limited. However, threading can still provide benefits when handling I/O-bound tasks, where threads can run in parallel and make better use of multi-core processors.

Creation and Management: Threading in Python is made straightforward through the built-in threading module. This module offers classes and functions for creating, starting, stopping, and managing threads.

Thread Safety: Threads can access shared resources, and this shared access can lead to race conditions and data corruption. Threading provides mechanisms like locks, semaphores, and conditions to ensure thread safety and proper synchronization.

Parallel Task Execution: Threading allows you to break down a larger task into smaller subtasks and execute them in parallel. This is particularly useful in scenarios where you have a set of similar tasks to perform concurrently.

Responsive User Interfaces: Threading is often used in graphical user interfaces (GUIs) to keep the user interface responsive while performing time-consuming operations in the background. This prevents the UI from freezing and improves the user experience.

Resource Sharing: Threads can share resources and data with each other, making it easier to pass information between different parts of your program. However, care must be taken to ensure data integrity through synchronization.

Deadlock Avoidance: Threading provides tools for managing potential deadlocks, a situation where threads wait indefinitely for each other to release resources. Techniques like timeout-based locks and deadlock detection are available for deadlock avoidance.

Distributed Computing: Threading can be used to implement distributed computing and parallel processing. It is often used in scenarios like web scraping, where multiple threads can fetch data from multiple sources simultaneously.

Event Handling: Threads can be used to handle events, such as responding to user input or reacting to changes in data, in a responsive and timely manner.

Load Balancing: Threading can be employed in load balancing scenarios where multiple threads are used to distribute and process tasks among different resources, such as CPU cores.

Task Prioritization: Threading allows you to assign priorities to threads, ensuring that critical tasks are executed before less important ones.

Example:

```
import threading
def print_numbers():
    for i in range(1, 6):
        print(f"Number {i}")
```

```
def print_letters():
    for letter in 'abcde':
        print(f"Letter {letter}")

# Create two threads
thread1 = threading.Thread(target=print_numbers)
thread2 = threading.Thread(target=print_letters)

# Start the threads
thread1.start()
thread2.start()

# Wait for both threads to finish
thread1.join()
thread2.join()

print("Both threads have finished.")
```

Synchronization:

Synchronization is the process of coordinating the activities of multiple threads to ensure that they access shared resources or data in a controlled and orderly manner. In a multithreaded environment, synchronization is essential to prevent issues like data corruption, race conditions, and deadlocks. Python offers several mechanisms for synchronization, including locks, semaphores, and condition variables.

Example:

```
import threading
total = 0
lock = threading.Lock()

def increment_total():
    global total
    for _ in range(1000000):
        with lock:
            total += 1
```

```
def decrement_total():
    global total
    for _ in range(1000000):
        with lock:
            total -= 1

thread1 = threading.Thread(target=increment_total)
thread2 = threading.Thread(target=decrement_total)

thread1.start()
thread2.start()

thread1.join()
thread2.join()

print("Final total:", total)
```

Multithreading in Python:

Multithreading in Python is the process of running multiple threads within a Python program. This enables concurrent execution and can significantly improve the performance of certain types of applications. Python's Global Interpreter Lock (GIL) can limit the true parallelism of threads, particularly in CPU-bound operations, but it's still beneficial for I/O-bound tasks.

1. **To do design the program for starting the thread in python.**

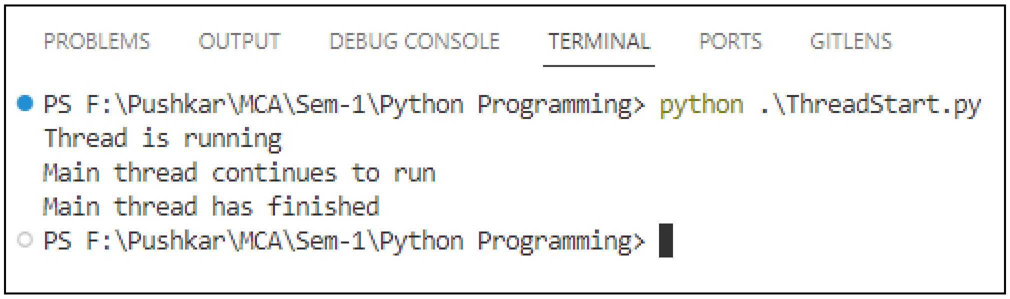
   Code:

```
import threading
# Define a function that will be executed in the thread
def my_thread_function():
    print("Thread is running")

# Create a thread object and pass the function to it
my_thread = threading.Thread(target=my_thread_function)

# Start the thread
my_thread.start()

# You can do other work here in the main thread
print("Main thread continues to run")

# Wait for the thread to finish (if needed)
my_thread.join()

# The program continues to execute here
print("Main thread has finished")
```

   Conclusion:

   The provided code showcases the fundamental principles of threading in Python. A new thread is created, and the my_thread_function is executed concurrently with the main thread. Threading allows for parallel execution, enabling the main thread to continue its tasks while the new thread performs its designated function. Proper synchronization with join() ensures orderly program termination. This simple example illustrates the power of threading in enhancing program efficiency and responsiveness by executing tasks concurrently.

Output:



```
PS F:\Pushkar\MCA\Sem-1\Python Programming> python .\ThreadStart.py
Thread is running
Main thread continues to run
Main thread has finished
PS F:\Pushkar\MCA\Sem-1\Python Programming>
```

2. **Write a program to illustrate the concept of Synchronization.**

Code:

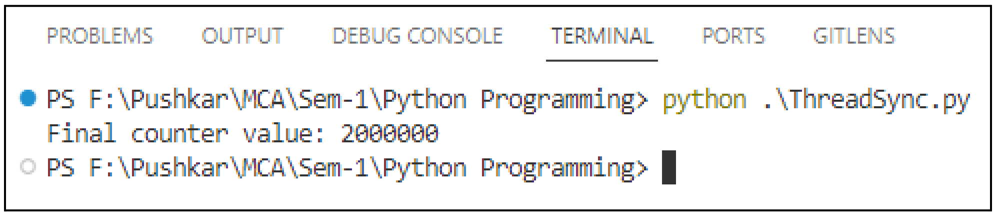
```
import threading
# Shared resource (a counter)
counter = 0

# Define a lock for synchronization
counter_lock = threading.Lock()

# Function to increment the counter safely
def increment_counter():
    global counter
    with counter_lock:
        for _ in range(1000000):
            counter += 1

# Create two threads to increment the counter
thread1 = threading.Thread(target=increment_counter)
thread2 = threading.Thread(target=increment_counter)

# Start both threads
thread1.start()
thread2.start()

# Wait for both threads to finish
```

```
thread1.join()
thread2.join()

# Print the final counter value
print("Final counter value:", counter)
```

Conclusion:

The provided code demonstrates the use of threading and synchronization to increment a shared counter safely. Two threads concurrently execute the increment_counter function, each adding a million to the counter. A lock (counter_lock) is used to ensure exclusive access to the shared resource, preventing race conditions. This example showcases the importance of synchronization in multithreaded applications to maintain data integrity. The final counter value reflects the successful coordination of the threads, emphasizing the power of threading and synchronization in concurrent programming.

Output:



```
PS F:\Pushkar\MCA\Sem-1\Python Programming> python .\ThreadSync.py
Final counter value: 2000000
PS F:\Pushkar\MCA\Sem-1\Python Programming>
```

3. **Write a program for creating a multithreaded priority queue.**

Code:

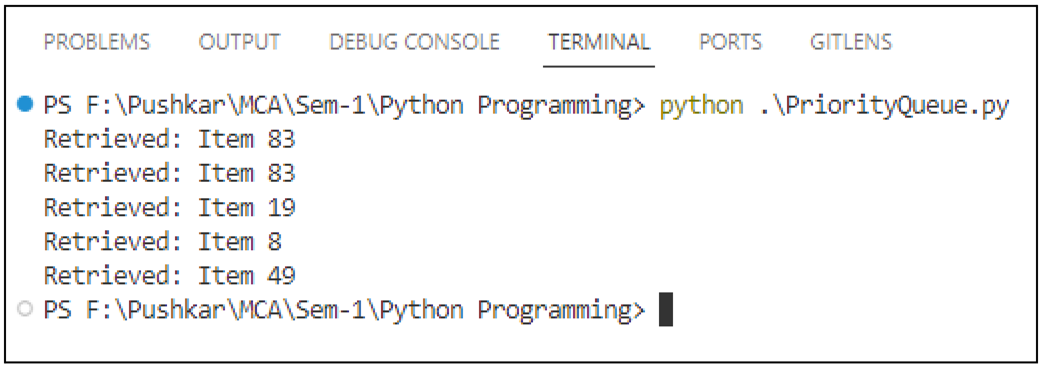
```
import queue
import threading
import time
import random

# Create a priority queue
priority_queue = queue.PriorityQueue()
# Create a flag to signal the retrieval thread to stop
stop_retrieval = threading.Event()
```

```
# Function to insert items into the priority queue
def insert_into_queue():
    for i in range(5):
        item = random.randint(1, 100)
        priority_queue.put((item, f"Item {item}"))
        time.sleep(1)

# Function to retrieve items from the priority queue
def retrieve_from_queue():
    while not stop_retrieval.is_set():
        try:
            item = priority_queue.get(timeout=1)
            print(f"Retrieved: {item[1]}")
            priority_queue.task_done()
        except queue.Empty:
            continue

# Create multiple threads for inserting and retrieving items
insert_thread = threading.Thread(target=insert_into_queue)
retrieve_thread = threading.Thread(target=retrieve_from_queue)

# Start the threads
insert_thread.start()
retrieve_thread.start()

# Wait for the insert_thread to finish
insert_thread.join()

# Set the stop_retrieval flag to signal the retrieve_thread to stop
stop_retrieval.set()

# Wait for the retrieve_thread to finish
retrieve_thread.join()
```

Conclusion:

The provided code demonstrates the use of the queue.PriorityQueue and threading in Python for concurrent data insertion and retrieval. An insertion thread adds random items with priorities to the priority queue, while a retrieval thread continuously retrieves and prints items. The code showcases the importance of synchronization and coordination in a multithreaded environment, as the insertion and retrieval threads work together. Additionally, the use of the threading.Event allows for graceful termination of the retrieval thread. This example highlights how Python's threading and queue modules can be used to manage shared resources and safely handle concurrent operations.

Output:



```
PS F:\Pushkar\MCA\Sem-1\Python Programming> python .\PriorityQueue.py
Retrieved: Item 83
Retrieved: Item 83
Retrieved: Item 19
Retrieved: Item 8
Retrieved: Item 49
PS F:\Pushkar\MCA\Sem-1\Python Programming>
```